Assessment Questions

1. Contrast RISC and CISC Architectures

Feature	RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
Instruction Set	Small, optimized instructions	Larger, more complex instructions
Execution Speed	Faster execution, as instructions are simple	Slower due to varied instruction lengths
Instruction Length	Uniform, typically fixed-length	Varies in size, often with complex opcodes
Hardware Complexity	Simple, requiring less hardware logic	More intricate hardware to decode complex instructions
Programming	Heavy reliance on compilers for optimization	Allows direct control with assembly programming
Examples	ARM, MIPS, SPARC	x86, Intel 8086

RISC is optimized for speed by simplifying instructions, while CISC focuses on reducing program size with more complex instructions.

---

2. What is the Role of the Program Counter (PC) in ARM Architecture?

The **Program Counter (PC)**:

- Points to the memory address of the next instruction.
- Automatically increments (typically `PC = PC + 4` for 32-bit instructions).
- Can be explicitly changed (e.g., during branch instructions).
- Crucial for program flow control, facilitating loops, function calls, and jumps.

---

3. Why Are Condition Codes Important in ARM Assembly?

Condition Codes:

- Flags set by operations like `CMP`, `ADD`, `SUB` to indicate the outcome of the operation.
- Key flags include:
  - **Z (Zero)**: Set if the result is zero.
  - **N (Negative)**: Set if the result is negative.
  - **C (Carry)**: Set if there is a carry or borrow.
  - **V (Overflow)**: Set if there is signed overflow.

Usage:

- Enable conditional execution of instructions (e.g., `EQ`, `NE`, `LT`).
- Enhance performance by reducing the need for branch instructions.

Example:

```
CMP R0, R1    ; Compare R0 and R1
BEQ equal     ; Branch if R0 equals R1
```

---

4. How Does ARM's Pipeline Architecture Enhance Performance?

The **ARM Pipeline** improves performance by processing multiple instructions simultaneously in stages:

1. **Fetch**: Retrieve the instruction.
2. **Decode**: Interpret the instruction.
3. **Execute**: Carry out the operation.

Benefits:

- **Higher Throughput**: Ideal pipeline conditions lead to one instruction per cycle.
- **Lower Latency**: Instructions are processed concurrently to minimize delays.

Advanced ARM processors use pipelines with superscalar and out-of-order execution to further optimize performance.

---

5. Explain Direct vs. Indirect Addressing Modes in ARM Assembly

Feature	Direct Addressing	Indirect Addressing
Definition	Operand's memory address is specified	Operand's address is in a register
Example	`LDR R0, =0x1000`	`LDR R0, [R1]`
Flexibility	Limited flexibility	More flexible due to pointer usage
Instruction Length	May include immediate values	Often shorter due to register use

Direct Addressing uses specific memory addresses, while **Indirect Addressing** refers to dynamic memory access using registers.

---

6. Distinguish Between LDR and LDM Instructions

Feature	LDR (Load Register)	LDM (Load Multiple)
Purpose	Loads a single register from memory.	Loads multiple registers from memory.
Usage	`LDR` fetches one register from a specific address.	`LDM` loads a group of registers starting from a base address.
Example	`LDR R0, [R1]`	`LDMIA R1!, {R0, R2, R3}`
Memory Access	Accesses one address for the operand.	Accesses a sequence of addresses starting at a base address.
Registers	Works with a single register.	Works with multiple registers.

`LDR` is used for single register loads, while `LDM` is used for loading several registers in one operation.

---

7. How Does the Stack Pointer (SP) Work in Subroutine Calls?

The **Stack Pointer (SP)** in ARM:

- Points to the top of the stack, where return addresses, local variables, and registers are stored.
- In **subroutine calls**, the return address is pushed onto the stack to resume execution after the function returns.
- The **stack pointer** adjusts (e.g., `SUB SP, SP, #4`) to allocate space for saved registers.
- The stack follows a **push-pop** mechanism:
  - **Push**: Store data, moving the `SP` downward.
  - **Pop**: Retrieve data, moving the `SP` upward.

Example of subroutine call:

```
    BL subroutine      ; Branch with Link
subroutine:
    PUSH {R0, R1, LR}  ; Save registers
    ; Subroutine work
    POP {R0, R1, PC}   ; Restore registers and return
```

---

8. How Are Interrupts Processed in ARM Architecture?

Interrupts in ARM are handled by the **Interrupt Controller** and involve the following steps:

1. **Interrupt Trigger**: External sources generate an interrupt.
2. **Interrupt Service Routine (ISR)**: Execution pauses, and control jumps to the ISR.
3. **Context Saving**: The processor saves its current state before handling the interrupt.
4. **Interrupt Masking**: ARM can prioritize and disable interrupts based on priority.
5. **Return from Interrupt**: After ISR execution, the processor restores the state and continues normal operations.

---

9. What Benefits Do Thumb Instructions Offer in ARM?

Thumb Instructions are a compact form of ARM's 32-bit instructions (16-bit), offering:

1. **Reduced Code Size**: Thumb instructions require less memory space, optimizing storage.
2. **Better Cache Efficiency**: More instructions fit in the cache, enhancing performance.
3. **Lower Power Consumption**: Smaller instructions reduce memory access and power usage, ideal for mobile or embedded systems.
4. **Performance Improvement**: In some cases, Thumb instructions provide faster execution due to efficient caching.

---

10. Swap Two Numbers Without a Third Variable in ARM Assembly

ARM assembly code to swap two numbers:

```
    LDR R0, =5          ; Load first number (5) into R0
    LDR R1, =10         ; Load second number (10) into R1

    ; Perform swap using arithmetic
    ADD R0, R0, R1      ; R0 = R0 + R1
    SUB R1, R0, R1      ; R1 = R0 - R1 (original value of R0)
    SUB R0, R0, R1      ; R0 = R0 - R1 (original value of R1)
```

After execution, `R0` holds 10, and `R1` holds 5, completing the swap.

---

11. What Is Endianness and Its Impact on ARM Memory?

Endianness describes the byte order in memory:

- **Big-endian**: Most significant byte stored at the lowest memory address.
- **Little-endian**: Least significant byte stored at the lowest memory address.

ARM processors support both endianness modes, with **little-endian** being more common. Mismanagement of endianness can lead to data corruption, especially when transferring data between systems using different endianness formats.

Example: In little-endian:

- `0x12345678` is stored as `78 56 34 12`.

In big-endian:

- `0x12345678` is stored as `12 34 56 78`.

---

12. What Does the Barrel Shifter in ARM Instructions Do?

The **barrel shifter** enables efficient bit shifting and rotation in ARM:

- **Logical Shifts**: Left or right shifts, filling with zeros.
- **Arithmetic Shifts**: Right shifts with sign extension.
- **Rotation**: Circular shifts that rotate bits around.

Example:

```
    LSL R0, R1, #4    ; Logical shift left: Shift R1 by 4 positions
    ROR R2, R3, #8    ; Rotate R3 by 8 positions
```

The barrel shifter allows fast bit manipulation without requiring extra instructions.

---

13. Why Is Pipelining Vital in ARM Processors?

Pipelining in ARM enhances performance by processing instructions in stages, with multiple instructions overlapping:

- **Increased Throughput**: Multiple instructions are processed in parallel.
- **Reduced Latency**: Minimizes idle processor cycles.

ARM processors typically have a 5-stage pipeline, enhancing efficiency, especially in embedded systems.

---

14. Difference Between Floating-Point and Integer Operations

Floating-Point Operations:

- Handle real numbers with fractions and wide ranges.
- May involve approximation due to limited precision.

Integer Operations:

- Handle whole numbers, ensuring exact calculations.

- ARM provides specific hardware support for both types.

---

15. Benefits of Inline Assembly in ARM C Programming

Inline Assembly enhances C programming by allowing low-level ARM-specific instructions:

1. **Performance Tuning**: Critical code can be optimized at the assembly level.
2. **Hardware Control**: Direct access to ARM features not available in C.
3. **Reduced Code Size**: Assembly often requires fewer bytes than C.
4. **System-Level Control**: Necessary for operations like interrupt handling and system register manipulation.

Example:

```
int a = 5, b = 10, result;
__asm__("ADD %0, %1, %2" : "=r" (result) : "r" (a), "r" (b));
```
